Aleksi Rasio

Integrating a hardware platform to a test automation solution

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Thesis

26 March 2018

Metropolia

Author	Aleksi Rasio
Title	Integrating a hardware platform to a test automation solution
Number of Pages	34 pages
Date	26 March 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Engineering
Instructor	Peter Hjort, Senior Lecturer

This Bachelor's Thesis documents a test automation project conducted in a Finnish ICT company. The objective of the project was to introduce a test automation solution for a consumer hardware platform and related software modules. Development of organizational software testing conventions and preparation for longer term requirements formed the basis for executing the project.

The primary success criterion for the project was to create a test automation system using Jenkins automation server software and Robot Framework, an open source test automation framework. The project begun by dividing the success criteria to smaller subtasks and by determining their interdependencies. First task was to ensure environment-agnostic test case execution. Ten test devices were then installed to a rack environment and network connectivity to each device was verified. The test device pool was configured to Jenkins as resources. Jenkins test jobs were then created to utilize the test devices. Select test jobs were also scheduled to execute hourly.

The solution was tested by employing it in daily software testing work. This revealed connectivity problems between the Jenkins server and test devices, which were rectified by introducing artificial latency to network calls. A memory leak issue on the Jenkins server was also identified during testing; it was fixed by introducing a missing command line parameter.

The resulting test automation system fulfilled all set requirements. The project and related process was very educational. The resulting system fulfilled the client's needs and is future-proof considering its extensibility and ease of modification.

Keywords	software testing, test automation, Jenkins, Robot Framework

Tekijä Otsikko Sivumäärä Aika	Aleksi Rasio Laitteistoalustan integrointi testiautomaatioratkaisuun 34 sivua 26.3.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Software Engineering
Ohjaaja	Lehtori Peter Hjort

Insinöörityö tehtiin suomalaisen tieto- ja viestintätekniikkayrityksen testiautomaatioprojektin yhteydessä, jonka tavoitteena oli tuoda kuluttajalaitteistoalusta ja siihen liittyvät ohjelmistomoduulit testiautomaation piiriin. Projektin taustalla olivat organisaation ohjelmistotestauskäytäntöjen muutokset ja valmistautuminen uusiin, pidemmän aikavälin vaatimuksiin.

Projektin onnistumisen kriteerinä oli luoda testiautomaatiojärjestelmä Jenkins-automaatiopalvelinohjelmistoa ja Robot Framework -testausohjelmistokehystä hyväksi käyttäen. Projekti aloitettiin jakamalla kriteerit pienempiin kokonaisuuksiin ja selvittämällä niiden keskinäiset riippuvuussuhteet. Ensimmäisenä työvaiheena oli testitapausten toiminnan varmistaminen suoritusympäristöstä riippumatta. Tämän jälkeen testilaitteita asennettiin räkkiympäristöön ja varmistettiin niiden saavutettavuus verkossa. Testilaitteet määriteltiin resursseiksi Jenkins-palvelimelle, jolle luotiin myös laitteita käyttäviä testaustyötehtäviä. Testiajoja myös ajastettiin suoritettavaksi tunnin välein, ja niiden toiminta varmistettiin palvelimen lokeista.

Järjestelmän toimintaa testattiin varsinaisessa ohjelmistotestaustyössä, joka paljasti yhteysongelmia Jenkins-palvelimen ja testilaitteiden välillä. Yhteysongelmat ratkaistiin lisäämällä viivettä palvelimen ja testilaitteiden välisiin verkkokutsuihin. Lisäksi havaittiin muistivuoto-ongelma, joka korjattiin lisäämällä yksittäinen puuttuva komentoriviparametri.

Projektin tuloksena syntynyttä testiautomaatiojärjestelmää hyödynnetään yrityksessä päivittäisessä ohjelmistotestaustyössä. Järjestelmä täytti tilaajan tarpeen ja on helposti muunneltavissa tulevaisuutta ajatellen. Projekti oli prosessina erittäin opettavainen.

Avainsanat	ohjelmistotestaus, testiautomaatio, Jenkins, Robot Framework

Contents

List of Abbreviations

1 Introduction 1

2 Software testing 1

2.1 Basis of software testing 2
2.2 History of software testing 3
2.3 Testing in agile software development 4
2.4 Time to market 5
2.5 Costs of software testing 6
2.6 Test automation 7
2.6.1 Robot Framework 9
2.6.2 Jenkins 10

3 Project 11

3.1 Client 12
3.2 Background 13
3.3 Product 13
3.4 Initial state 14
3.5 Implementation 14
3.5.1 Test case generalization 15
3.5.2 Test device rack setup 15
3.5.3 Jenkins configuration 17
3.6 MVP reception 20
3.7 Stabilization 20
3.8 Future improvement 26

4 Conclusion 28

References 31

List of Abbreviations

API	Application programming interface. Set of predefined methods for communication between software components.
ASCII	American Standard Code for Information Interchange. Character encoding standard for digital communication.
HTML	Hypertext Markup Language. Syntactic annotation language for creating web pages.
HTTP	Hypertext Transfer Protocol. Application protocol for request-response based data communication.
IEEE	Institute of Electrical and Electronics Engineers. IEEE is the world's largest technical professional association, comprised of over 395 000 members in 160 countries.
IP	Internet Protocol. Primary communications protocol of the Internet protocol suite, utilised for relaying datagrams.
MVP	Minimum viable product. A product with a minimum set of features that satisfies its use case.
RAM	Random-access memory. Data storage which is used for storing currently used machine code and data. Near-equal speed of read and write operations is a distinctive trait of random-access memory.
SDK	Software development kit. A set of tools and documentation for developing software for a specific target platform.
XML	Extensible Markup Language. Syntactic language for encoding documents in human-readable and machine-readable format.

1 Introduction

This Bachelor's Thesis studies software testing and test automation and documents a test automation project conducted at Elisa Oyj, a Finnish telecommunications company. Test automation refers to the process of automating software testing.

The project consisted of setting up a Robot Framework and Jenkins -based test automation system, configuring a test device pool and iterating the setup further to stabilize its operation. In-house developed software modules were the primary focus of testing, but the performance of the device platform was a subject of interest as well.

Requirements for a *minimum viable product* (MVP) were defined by the client to assess the project outcome: they are described in detail in paragraph 3.3. The criteria were part of an organizational endeavour to establish a common test automation practice. This Bachelor's Thesis covers project stages up to reaching the MVP status and subsequent system stabilization efforts.

Chapter two serves as an introduction to the concept of software testing: its practical and financial motivation, history and automation are explored. Chapter three documents efforts of the client project. Elisa Oyj, its organizational structure, and financial figures are documented. Project background and motivation are described before documenting the stages of its execution. Chapter four is a summary and conclusion of the study.

2 Software testing

IEEE standard on software testing (29119-1) defines that creating flawless software is generally acknowledged as an impossible task. To reduce risks and errors harmful for the user experience, it is necessary to test software prior to its delivery. [1, p. 13.]

Software testing can be defined as systematic process of finding errors from a software product. Software specifications, upon which the software product has been developed, are used as the basis of the process. Inputs, outputs and correct results of the tested entity are defined from the specifications. [2, p. 10–11.] It should be noted that comprehensive testing of all software functionality is generally unfeasible, especially in the case

of complex software products [1, p. 14]. Cost-effective software testing practices are discussed in paragraph 2.5.

Software development is the only engineering discipline in which product testing is a major technical and organizational challenge and an important cost factor. The impact of this phenomenon is product-specific and depends on three primary factors:

1. scope and complexity of the software product
2. lack of general standards for the software development process, resulting in the inability of assuring product quality by examining the development process
3. scarcity of practical and scalable methods for measuring software product quality through static analysis, i.e. the process of testing program code without executing it.

Changes introduced to product specifications during development and maintenance processes also affect product testing. This is a distinctive trait of modern software development. Traditional industry manufacturing processes rely on up-front engineering work to deliver product specifications, which are presumed to be final and not subjected to major further changes. In the case of software products, specifications can change considerably during development, especially when using agile methodology described in paragraph 2.3. [1, p. 6; 3, p. 14.]

2.1 Basis of software testing

Primary motivation for software testing is to assess the functional quality of a software product. Resulting evaluation is used as a basis of fixing defects within the product. Performance, reliability and security can be applied as indicators of functional quality. [3, p. 130.] By assessing product quality, the number of software execution errors visible to the customer can be reduced, thus avoiding possible damages to customer relations and company finances [4, p. 1; 5, p. 8]. It can also be argued that expectations for performance and quality of software products have grown, given the widespread usage of software [11].

Software testing is becoming more valued in software development projects, as it has proven to be an effective tool for increasing product quality. Testing itself does not guarantee product quality but can be employed alongside prime design and quality-centric development practices, such as code reviews. The benefit of software testing correlates

with the relative starting time of testing within a project: if errors are detected soon, less working time is needed for fixing them (see figure 1). The principal basis of software testing can thus be viewed as essentially financial. [6, p. 29–30.]

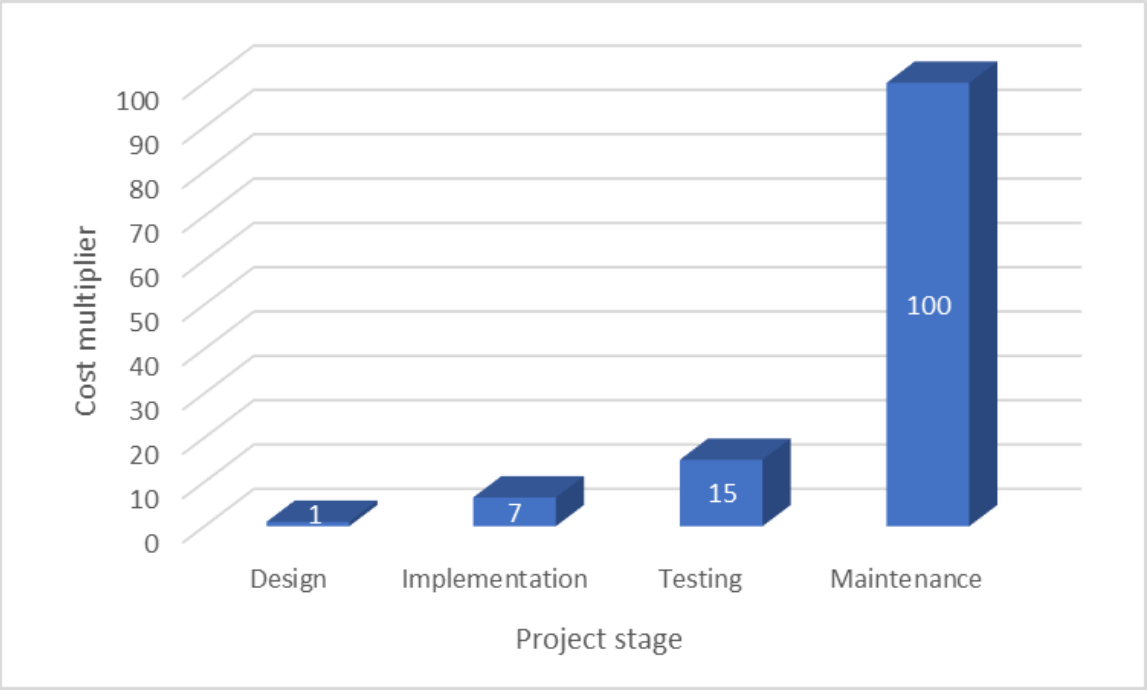

Figure 1. Cost of fixing a software defect relative to ongoing project phase [7.]

2.2 History of software testing

Software testing was not yet recognized as an independent process within the emerging software industry in the early 1950's. Testing was viewed as a part of the debugging process, but the concept of software testing slowly began to take shape by the end of the decade. [8, p. 23.] Software products at that time were generally quite simple and could be comprehensively tested with a limited set of test scenarios. Testing was carried out with pen and paper and consisted primarily of proving the correctness of algorithms within the software. [2, p. 12.]

The amount of software products and their prices rose during the 1960's. More complex software was tested, but the rigidity of development and related tools caused temporal challenges in fixing defects. Software products were often delivered consciously in a faulty state if there was no time to carry out fixes.

The time span from 1970's to the end of the 1980's was a crucial period for the software industry: programming work and software development processes evolved more systematic. Software testing was defined as methodical execution of a software product with the purpose of detecting errors. First testing and test automation tools were released in the 1980's. During these two decades, the aspiration of software testing changed decisively: the objective was no longer to prove the impeccability of software, but to reveal its errors.

In the 1990's, the popularity of object-oriented programming languages soared. This caused new challenges for software testing in terms of object inheritance, state management and interface errors. Development of software testing tools and test automation was invested upon as the number of test scenarios increased. [8, p. 23–25.]

2.3 Testing in agile software development

Within projects following the linear waterfall model (see figure 2), it is common that testing the software product begins in the final stages of the project. The role of software testing in the waterfall model resembles a final product inspection employed within traditional manufacturing industry.

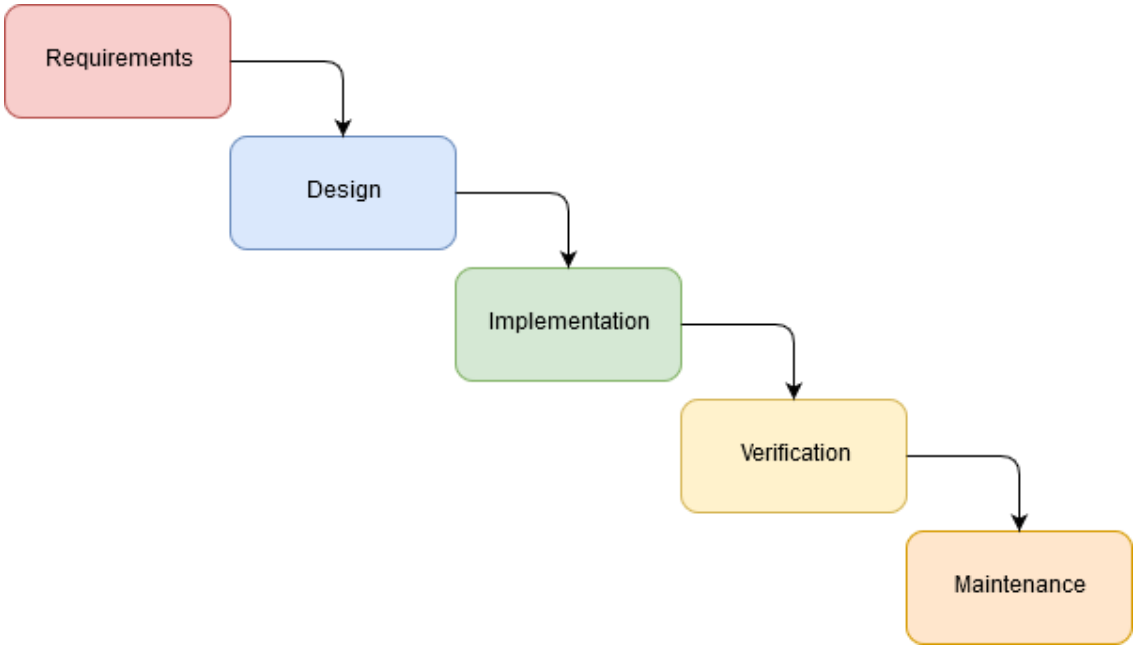

Figure 2. Waterfall model [9, p. 2].

Migration to agile development methodology has changed the nature of software testing. In agile methodology, software development is carried out iteratively (see figure 3) in short, consecutive periods. The aim is to possess the capability to deploy software continuously to a production environment, i.e. available for use of actual customers and users. It is therefore necessary to carry out testing continuously, along with development work. [6, p. 30.]

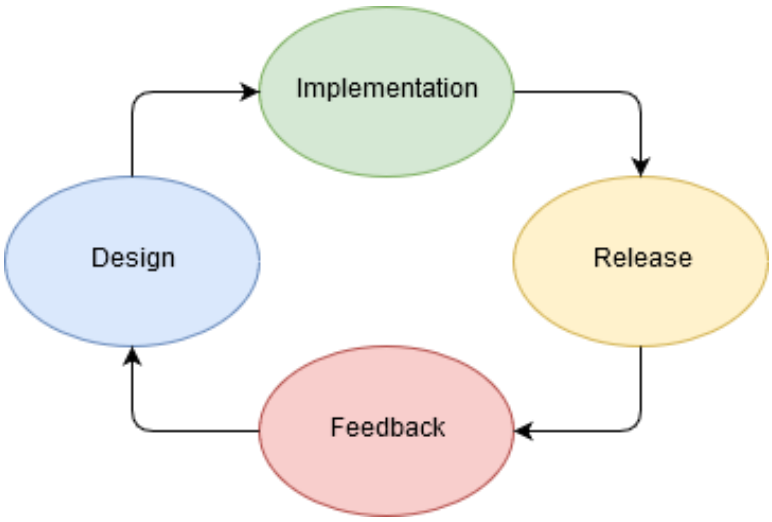

Figure 3. Agile development iteration [9, p. 3].

2.4 Time to market

The extent of software testing directly affects the product's release date, or *time to market*, which denotes the length of development time required to ship a product. Test data is generally used in assessing if a product is ready for the market.

Traditional quality assurance cycles spanning multiple weeks are generally unacceptable, as organizations face pressure to release products as quickly as possible. This can lead to dilemma of having to pick between product quality and shipping speed. Agile development and its requirement of constant testing can however resolve this conundrum, as it necessitates that testing progresses at the speed of development. [11.]

2.5 Costs of software testing

As stated in paragraph 2.1, the principal motivation for testing a commercial software product is financial. Average overall costs of contemporary software testing are approximately 26 % of the project budget [5, p. 11]. As each software product and related project is profoundly different, there is no generally accepted rule for budgeting software testing.

Prerequisites for cost-effective software testing are employing a risk-based testing model and defining test coverage requirements [1, p. 24; 5, p. 42–43]. Risk-based testing means targeting tests to product functionality deemed most critical for customers, based on a risk analysis [1, p. 24; 6, p. 35]. Test coverage denotes the ratio of code statements covered by test scenarios compared to the whole code base [2, p. 52]. By defining the coverage requirements, excessive testing can be avoided. Constraints should be set due to the imperfect nature of software products: with complex products, minor defects can be discovered nearly endlessly, leading to escalation of the product's time to market if they would all be assessed. [10.]

Figure 4 illustrates the ratio between financial investment to software testing and the number of errors in delivered software. The interception point of the two curves denotes the most cost-efficient investment.

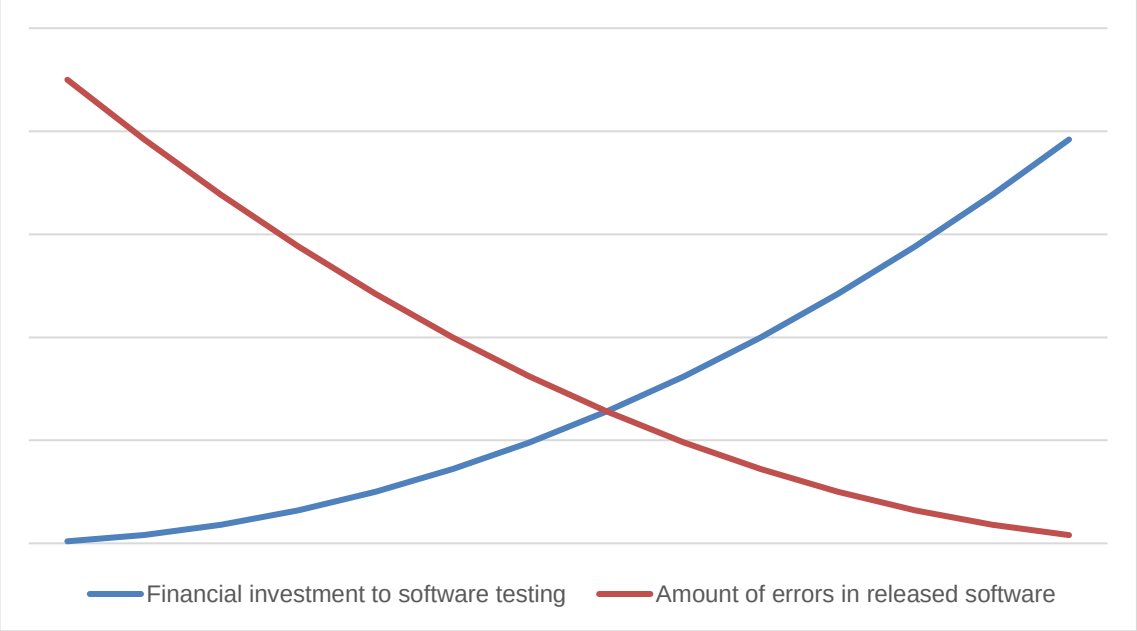

Figure 4. Ratio of financial investment and the amount of errors in released software product [10].

Traditionally, the single most expensive cost of software quality assurance has been the overall cost of human resources used for testing [5]. This has however changed in recent years, as exhibited in figure 5. Cost-efficiency of testing can be improved with test automation: it enhances human resources -wise test work effectiveness and shortens the time to market. [8, p. 52; 11.] Continuous testing required by agile development methodology is practically viable only through test automation [6, p. 30].

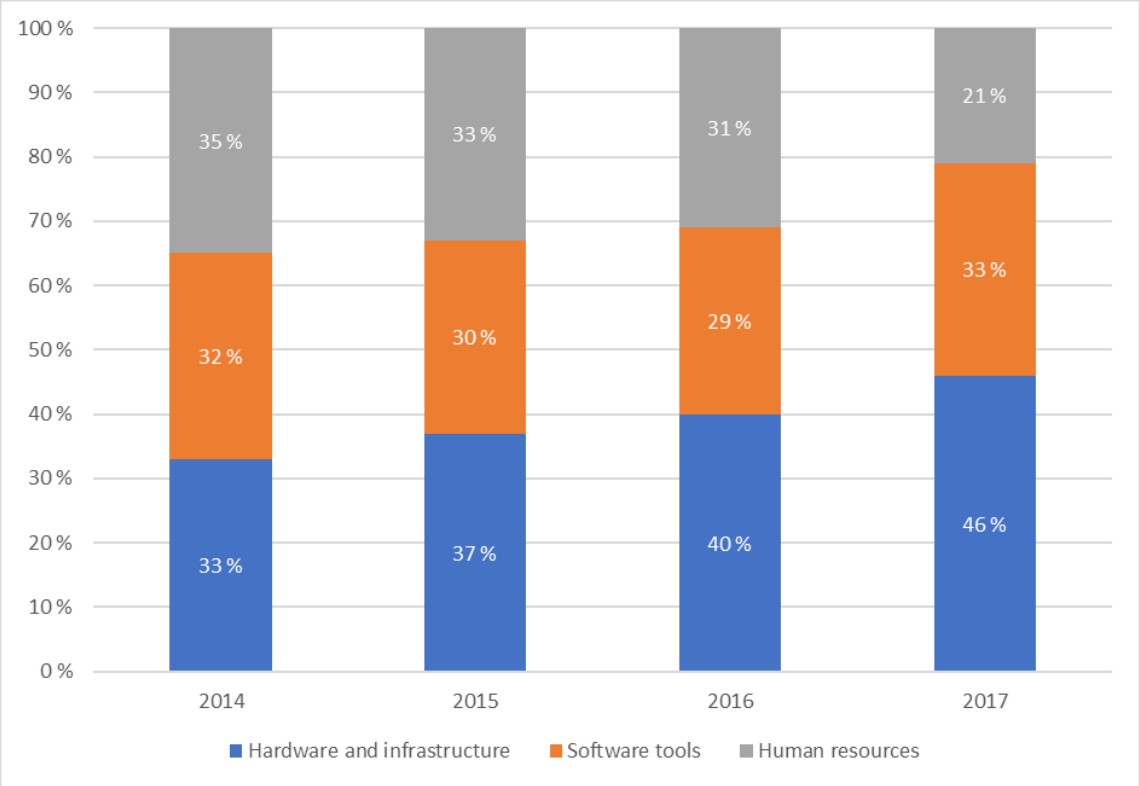

Figure 5. Development of quality assurance budget composition from 2014 to 2017 [11].

2.6 Test automation

Software test automation comprises any automated tools that examine software for errors. A popular subset of test automation tools is programming language -specific automated static analysis functionality within integrated development environments, which continuously scans program code for errors. Integrated development environments, *IDEs*, are software packages that provide thorough facilities for developing software.

The term *test automation* is however commonly used to refer to applications and scripts which analyse and evaluate program execution dynamically, i.e. during runtime. [6, s.

30–31.] Benefits of test automation, as reported by organizations that have adopted such practices, are exhibited in figure 6.

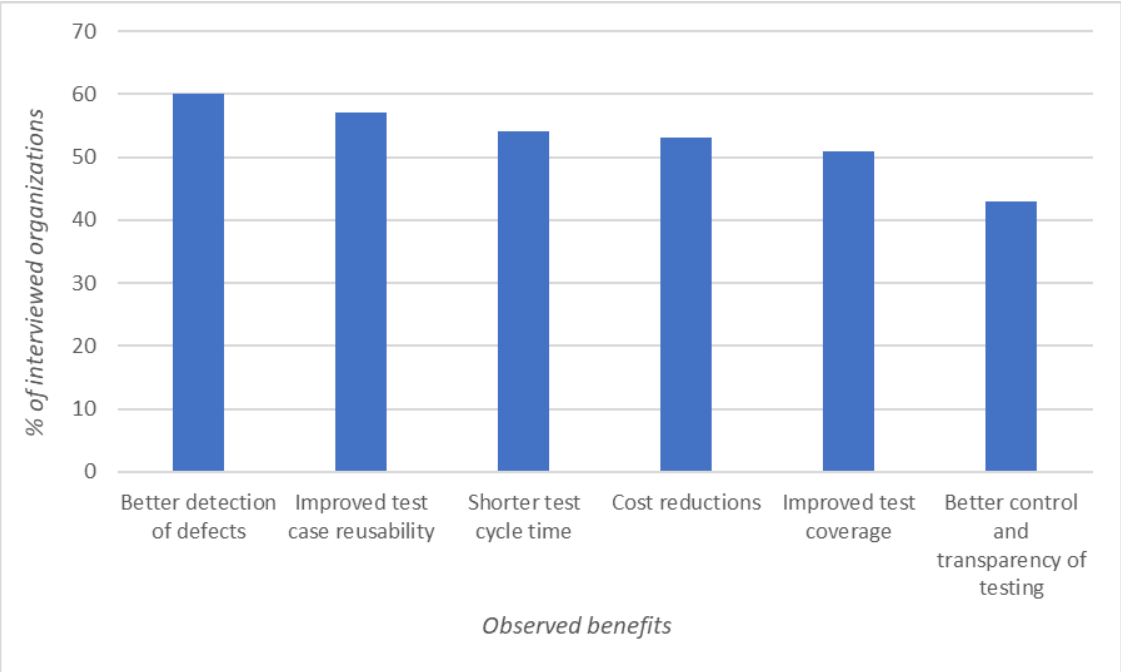

Figure 6. Benefits of software test automation reported by organizations [5].

Test automation tools are divided to:

- in-house solutions developed by organizations for their own exclusive use
- open source software solutions
- commercial proprietary-licensed software.

Internal tools' specifications and development can be thoroughly controlled, but costs of development and maintenance are cast on the organization itself. Commercial solutions are developed and maintained by external organizations but acquiring such products can be costly and the acquisition binds the organization to a third party. Open source projects are often functionally sufficient and their liberal licensing guarantees that the acquisition does not result in any costs. Open source projects may however be lacking in regular maintenance and formal on-demand technical support is unavailable. [12.]

Software test automation is becoming more popular, but only 15–16 % of all test work is automated [5, p. 28]. Issues in adopting test automation reported by organizations are for example:

- lack of mobile device support within the automation solution
- availability issues with the test environment and related test data
- lack of an appropriate test automation process
- lack of test automation tools
- integration problems among separate test automation tools
- shortage of personnel competent in test automation.

Because of these issues, it is rare for an organization to have automation employed across the whole testing process: instead, a specific part of the process is automated, such as regression testing. [5, p. 28.] Regression testing means re-testing a previously tested component after it has been modified. Regression testing is performed to verify new implementations have not affected the software's unmodified functionality. [1, p. 29.]

Following subparagraphs, 2.6.1 and 2.6.2, study two popular tools as exemplary cases of test automation software.

2.6.1 Robot Framework

Robot Framework is an open source test automation framework. It is operating system independent and application-agnostic. [13.] Robot Framework is based on Pekka Klärck's master's thesis approved in 2005, titled "Data-Driven and Keyword-Driven Test Automation Frameworks" [14].

Robot Framework was initially developed at Nokia Networks as an in-house tool, but it has been licensed as open source software since the release of version 2.0 in 2008. Development of Robot Framework is sponsored by the non-profit Robot Framework Foundation. [13; 15.]

Test cases for Robot Framework are written in Python programming language. Test cases can be combined to larger test suites and they are executed with a command-line tool. After test execution, Robot Framework creates HTML-formatted test reports and logs (see image 1) along with an XML-formatted test log file. The XML-file and presence of command-line tools enable scripting and post-processing of Robot Framework, ultimately allowing its integration as part of a larger automation pipeline. [16.]

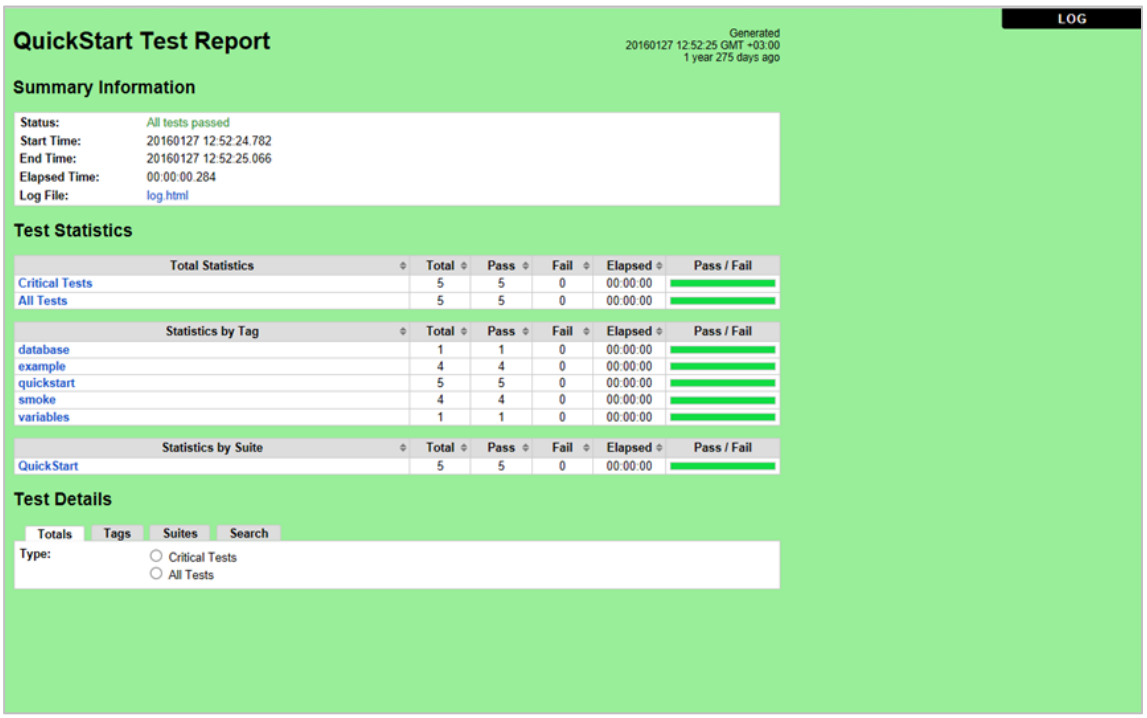

Image 1. HTML test report created by Robot Framework [13].

The open source Apache License 2.0 -license used by Robot Framework allows modification of the entire framework. Most libraries and tools available for Robot Framework are also licensed as open source projects. [13.]

2.6.2 Jenkins

Jenkins is an open-source cross-platform automation server. The software is distributed in many diverse packages, including a standalone Java WAR archive, a Docker image and various operating system -specific installers. Typical use cases for a Jenkins server include building software projects, running test automation and carrying out software deployments. Jenkins scales well from automating simple tasks to constructing complex pipelines with multiple stages (see image 2). [17].

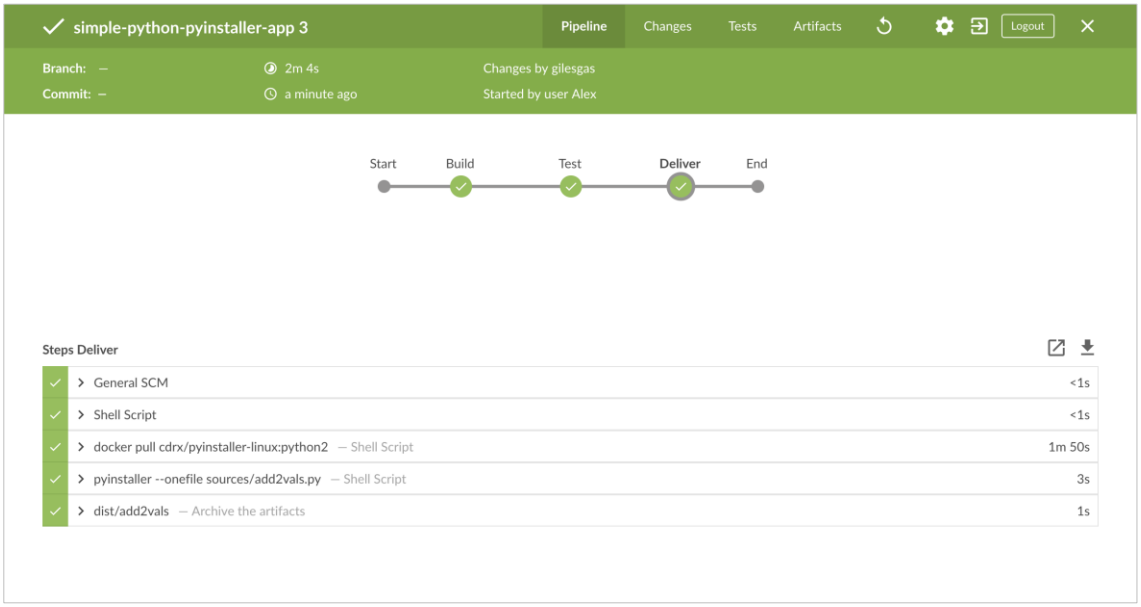

Image 2. Successfully finished pipeline build in Jenkins' Blue Ocean interface [18.]

Jenkins has an active development and user community, with over 1400 plugins available to suit various usage scenarios. Although Jenkins' parent project, Hudson, was originally developed at Sun Microsystems for continuous integration and delivery of Java code, Jenkins can be used to automate a wide array of tasks, regardless of programming languages used within a project. [17].

3 Project

The project documented in this thesis was carried out during the author's Software Developer traineeship with Elisa Oyj in 2017. Timeline of the project spanned from June to December 2017.

The project of the project was to study a process of integrating a hardware device to a test automation solution. The device was to serve as a platform for executing test automation suites against in-house developed software modules.

3.1 Client

The client for the innovation project was Elisa Oyj, a Finnish telecommunications, ICT and online service company. Elisa serves approximately 2,3 million consumer-, corporate- and public administration customers, which makes Elisa the market leader in its field in Finland. The company is listed on Nasdaq Helsinki Large Cap and its revenue in 2016 was 1,64 billion euros. Elisa has approximately 200 000 shareholders and 4300 employees. [19.]

Elisa's business operation is divided to Consumer and Corporate Customers units, illustrated in figure 7. These units are served by the Production unit and various support functions. [20.]

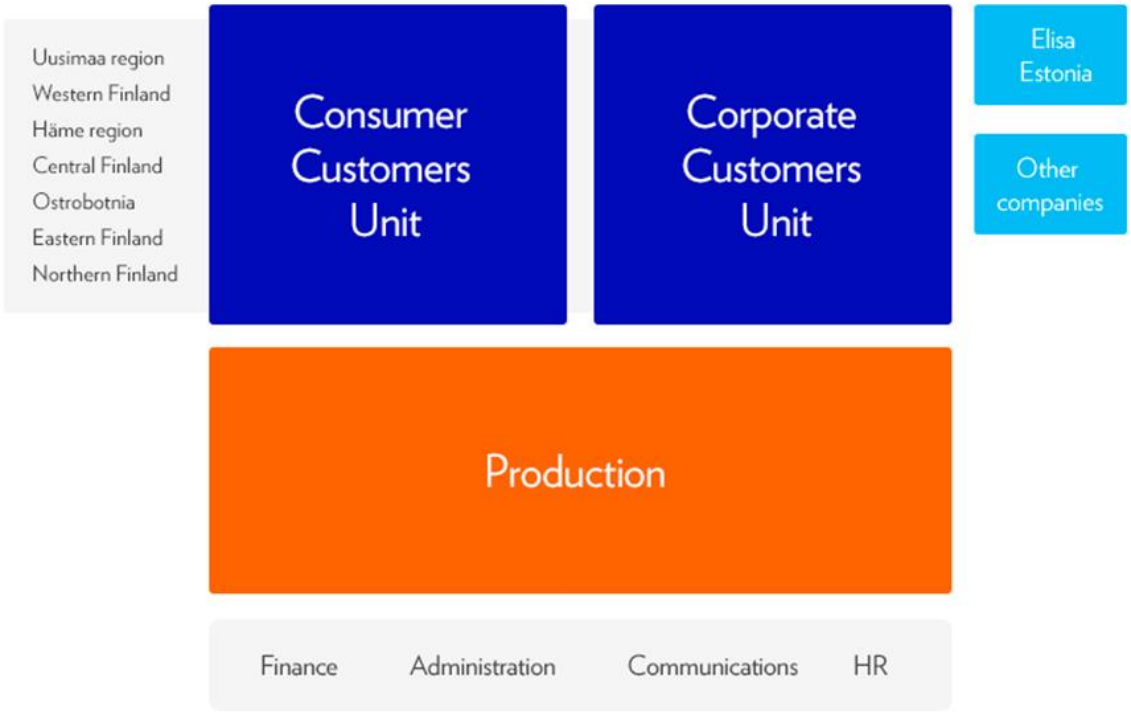

Figure 7. Operational model of Elisa Oyj [20.]

The author's position in Elisa's organization was within a development team in the Software Services subunit of Production.

3.2 Background

A common roadmap for the department was revealed in June 2017, setting following requirements for development teams:

- further development of Robot Framework -based test automation suites for software modules
- extend the usage of dedicated Jenkins-based automation servers
- migrate Robot Framework test suites as build jobs to the Jenkins environment
- automatically execute jobs against developed software on a regular schedule.

After assessing equipment and human resources available to the team, it was determined that all prerequisites were met for starting a project to fulfil new test automation requirements.

3.3 Product

The team was responsible for developing software modules and related APIs for a proprietary hardware device platform. Devices were deployed to consumer end-users who were billed periodically for service subscription. Various hardware revisions and successive models of the device family were deployed to customers simultaneously. All models supported same firmware releases and were compatible with each other as a device family.

The platform exposed a middleware with ECMAScript 5 -compatible JavaScript environment. Proprietary nature of the platform set a unique challenge for development. Explicit knowledge sharing with other developers was not possible, contrary to, for example, open source frameworks.

The MVP criteria for the test automation project was set as follows:

- A test device of each hardware revision is available for Jenkins for test suite execution
- Latest test suite revisions are automatically fetched from Git-repositories by Jenkins

- A brief, business-critical test suite is run on regular intervals against two separate software modules to verify and monitor correct operation of software modules executing on the device platform.

APIs and related server systems had their own separate testing and monitoring schemes, and thus excluded from the project's scope.

3.4 Initial state

It was assessed that the project would not to require full contribution of all primary team members, therefore a smaller project team was established. The primary team's Test Developer was tasked with overseeing the project and the author volunteered to join the project team.

Existing resources available for the team were examined during initial project planning. It was determined that all required equipment and software to start the project was already available.

Devices of all currently deployed revisions and models were available and a sufficient number was reserved explicitly for testing purposes. A standard 19" device rack from a previous testing endeavour was available for housing the test devices. The rack had pre-existing network connectivity via a router with an ample number of Ethernet cables. A server running Jenkins automation software on CentOS Linux distribution was also available, but it was not yet configured for the team's specific test automation needs.

3.5 Implementation

After equipment assessment, a project roadmap was drafted. Three separate tasks could be identified to fulfil MVP criteria:

- environment-wise generalization of Robot Framework tests
- setting up test devices to the equipment rack
- Jenkins server configuration.

The tasks were deemed to be joint efforts and both team members would work on all of them. Gaining comprehensive experience in a project of this nature was agreed to be

beneficial for team members' professional competence by both project staff and management.

3.5.1 Test case generalization

Proper execution of tests in a test automation server environment requires environment-agnostic test setup. This means that the tests must be self-contained and not reliant on any manual preparation of the environment. To test the generality of Robot Framework test cases, they were cloned from a Git repository to a personal workstation without any prior test configuration.

The repository was deemed well-generalized, as it utilized a Docker container for executing Robot Framework. Docker containers are lightweight stand-alone images containing all dependencies required to execute enclosed software [21].

To enforce information security, devices required manual configuration for each associated workstation. This step was crucial from a security viewpoint and did not affect test case generalization per se. Device configuration was a brief, one-time manual task per workstation and did not require considerable effort.

3.5.2 Test device rack setup

A total of 10 test devices of different hardware revisions were accumulated to form an initial device pool. Prerequisites for successful device testing were following:

- device settings had been properly configured on the workstation executing the tests
- device had an active service subscription
- device had network connectivity and reachable from the workstation.

Correct configuration was verified individually for each device. Proper device operation was finally verified with brief manual testing of deployed software modules. This step ensured the device was ready for integration to the rack.

Once device pool candidates were ready for installation to the equipment rack, a networking scheme was planned. An IP host address range .200–.254 was available on an existing test device network, supporting a maximum of 55 test devices. The number of

available host addresses exceeded needs for the MVP and future-proofed integration of further devices to the test automation solution.

Each test device was individually configured to use static IP addresses as listed in table 2.

Device model	Hardware revision	IP
A	v1	.211
A	v1	.212
A	v1	.213
A	v2	.214
A	v2	.215
A	v2	.216
A	v3	.217
A	v3	.218
B	v1	.231
B	v1	.232
B	v1	.233

Table 1. IP mapping of test devices.

A gap in IP addresses was left between the two hardware models to allow adding older devices while retaining logical sequencing. Vacant IP addresses were left to the beginning of the range to accommodate possible prototype- and other temporary devices.

After the test devices were configured and verified to be operating correctly, they were installed to the device rack. Each device was finally tested for basic network connectivity by pinging their respective IP addresses remotely. *Ping* is a network utility that can be used to test if a remote host is reachable and responsive. Final device rack setup is exhibited in figure 8.

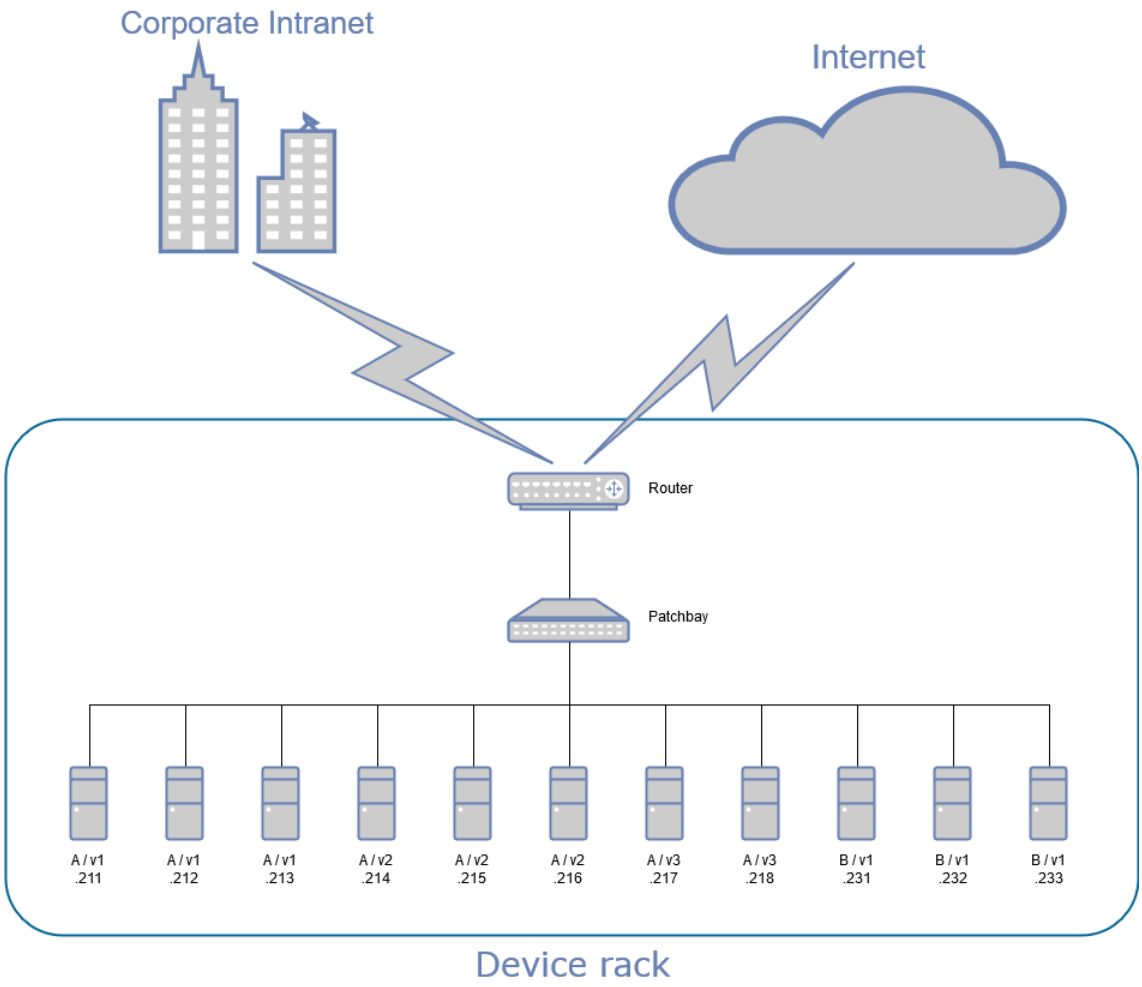

Figure 8. Final device rack configuration.

3.5.3 Jenkins configuration

Devices were now operational and ready to be configured as a device pool to the Jenkins automation server. Optimal approach for introducing the devices as assets to Jenkins was studied carefully. Lockable Resources plugin for Jenkins, pictured in image 3, was chosen as the most suitable way of compiling a test device pool.

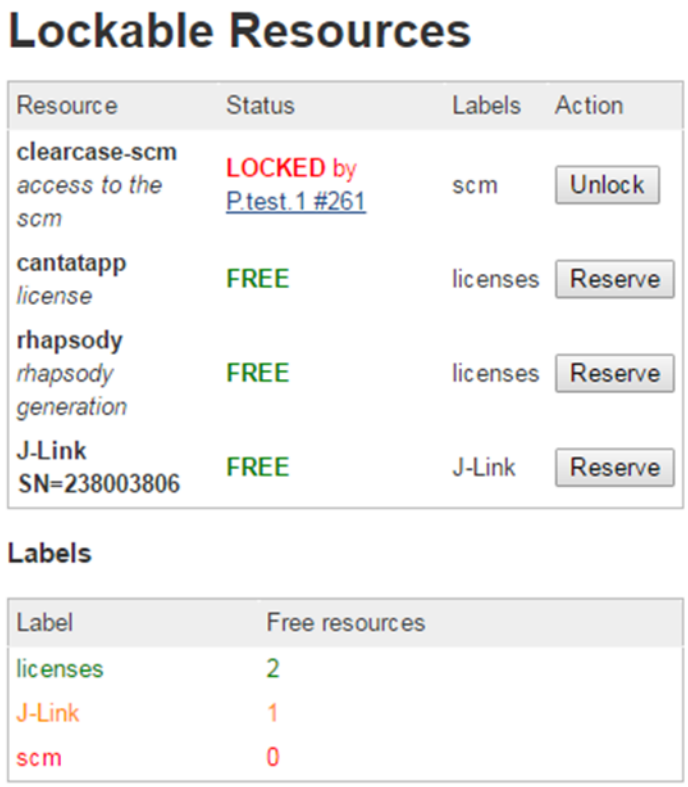

Image 3. Lockable Resources plugin for Jenkins [22].

Lockable Resources plugin fit the team's needs as it allowed setting up devices that are exclusively reserved for a specific job for its duration. The nature of the device platform's framework was single-tasking in the sense that a single end-user application could be executed at a time. This meant that a dedicated device was required to be fully committed, i.e. locked, for executing a single software test job.

Devices were configured with the Lockable Resources plugin under various tags based on their hardware revisions as listed in table 3. These explicit tags allowed targeting tests jobs against different device revisions or executing tests device-agnostically by using the Any-Device -tag.

Devices		*Tags*				
Device model	**Hardware revision**	**Any-Device**	**Device-A-v1**	**Device-A-v2**	**Device-A-v3**	**Device-B-v1**
A	v1	x	x			
A	v1	x	x			
A	v1	x	x			
A	v2	x		x		
A	v2	x		x		
A	v2	x		x		
A	v3	x			x	
A	v3	x			x	
B	v1	x				x
B	v1	x				x
B	v1	x				x

Table 2. Mapping of tags for test devices within Jenkins Lockable Resources plugin.

As test devices were successfully configured within the plugin, Jenkins test jobs were needed to properly utilize them.

Basic business-critical test jobs were created under the Any-Device -tag, since they should execute successfully on any device configuration. The jobs were configured to execute following sequence of subtasks:

- Clear the workspace.
- Clone master-branch of a Git repository containing Robot Framework tests to workspace root.
- Execute desired test suite.
- Publish the test suite report on its completion via Robot Framework Plugin for Jenkins, as demonstrated in image 4.

Tests were triggered manually during development, but after their rudimentary operation was verified to operate correctly, they were scheduled to run on an hourly basis.

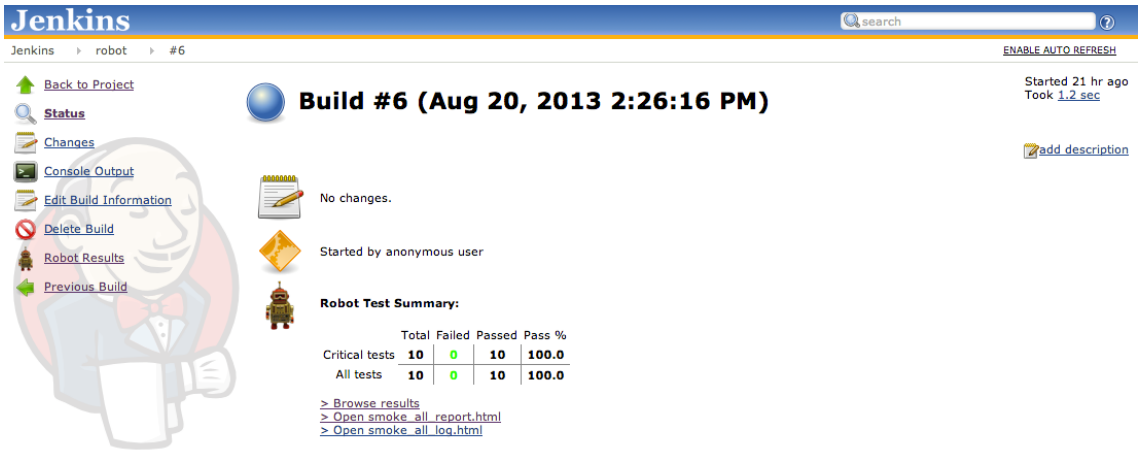

Image 4. Jenkins build page with Robot Framework results [23.]

MVP requirements for test automation were met after the tests were verified to be executing autonomously on a set schedule. Remarks on the operation of the system are detailed in paragraph 5.

3.6 MVP reception

MVP deliverable of the test automation project was finished mid-October 2017. The team's achievements were applauded by the management, with special merit given for swift implementation of the test automation solution. All requirements for the MVP were reached.

Further development of the test automation system was strongly encouraged to set an example to other teams tasked with fulfilling similar requirements. Post-MVP stage consisted of using the system in actual software testing work. This process proved effective for revealing impairments within the system.

3.7 Stabilization

Day-to-day automated software testing work was carried out to test the system itself. It was quickly noted that there were stability issues related to HTTP method calls between the test server and the test device's built-in REST API. The test device REST API was

the most critical part of the testing scheme, because it served internal state data of applications being executed (see figure 9). Returned data was compared to expected data defined in Robot Framework test scenarios.

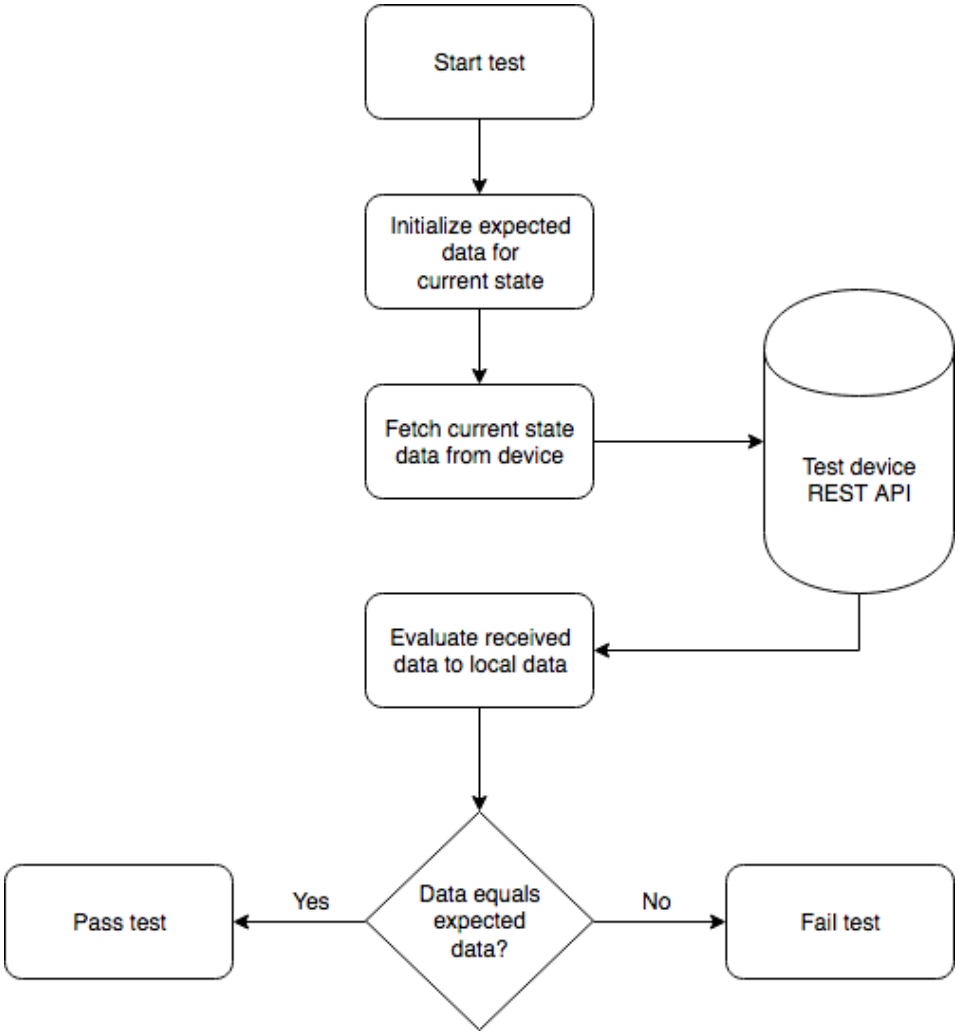

Figure 9. Common test case flow.

The errors were initially detected after multiple, seemingly random failures in tests that had been previously verified to finish successfully in a local workstation environment. Robot Framework logs were investigated, and all such failures exhibited a similar error message (see listing 1).

```
[ WARN ] Retrying (Retry(total=2, connect=None, read=None, redirect=None, sta-
tus=None)) after connection broken by 'ProtocolError('Connection aborted.',
BadStatusLine("''",))'
```

Listing 1. Terminal output of a protocol error occurred test execution.

Tests were thoroughly rerun locally to determine whether the problem originated from the rack network environment. The error was verified to occur also locally, but also in a similarly random pattern: no clear reason for the failure could be deciphered.

Documentation of urllib3 Python HTTP client used by the Robot Framework environment was studied and a single, brief description of the exception was found within HTTPConnectionPool class constructor parameter reference:

> **strict** – Causes BadStatusLine to be raised if the status line can't be parsed as a valid HTTP/1.0 or 1.1 status line, passed into httplib.HTTPConnection [24.]

The exception description hinted that the test device's internal HTTP server could be exhibiting non-standard behaviour. The concept of status line is identical in both HTTP/1.0 & HTTP/1.1 protocol specifications. Status line is the first line of a HTTP response message consisting of following, ordered keywords:

1. Protocol version – "HTTP/" followed by version specifier, for example "1.0" or "1.1", terminated with US-ASCII SP -whitespace character.
2. Status code – Three-digit status code, for example "200", terminated with US-ASCII SP -whitespace character.
3. Reason-phrase – Description of the status code, for example "OK", terminated with sequential CR and LF characters (US-ASCII CR carriage return and US-ASCII LF linefeed, respectively). [25; 26].

Network traffic was captured with Wireshark packet analyser software during local test execution to learn the exact format of the faulty status line (see image 5). A faulty packet was captured after several passes and it verified the BadStatusLine -exception message exhibited in listing 1: the response status line was missing altogether.

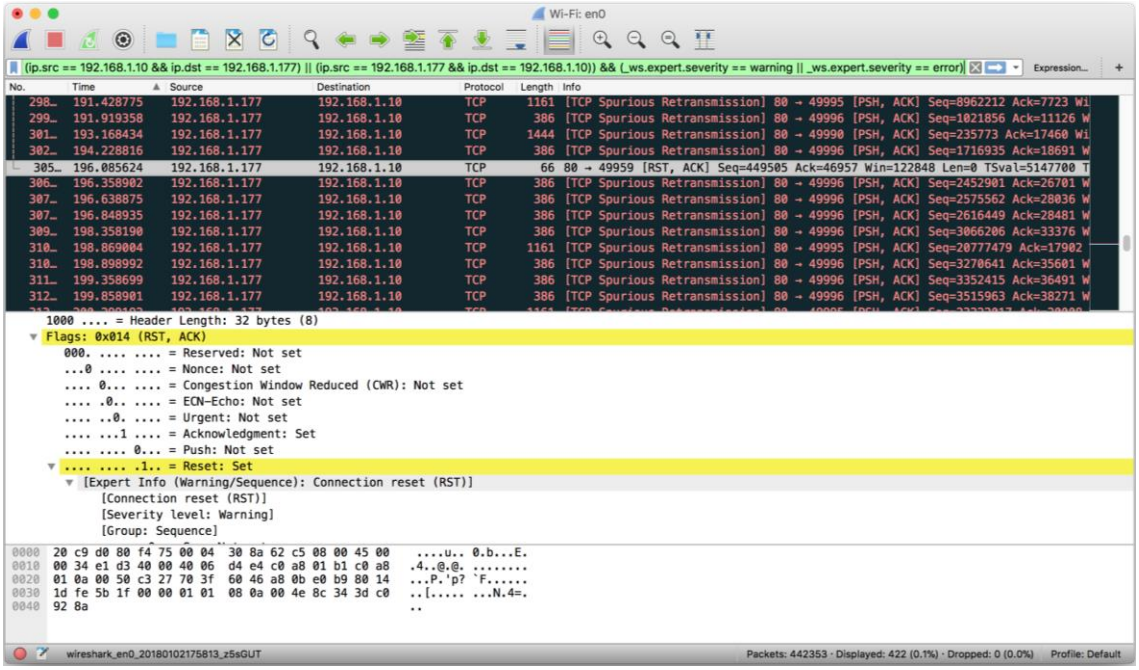

Image 5. Inspecting erroneous packets captured in Wireshark.

The packet originated from the test device's internal HTTP server and it importantly pointed out a general stability issue in the test suites. Considering the upcoming roadmap requirements for automatic alerts, a test case should be able to recover from minor network errors to prevent false positives from being reported. Improved network error handling was to be assessed.

It was suspected that the HTTP session initialized between the automation server and target device did not tolerate connection errors, possibly due to an insufficient number of retries defined in common test suite configuration. HTTP session creation in Robot Framework test cases was handled using the Create Session -keyword of *robotframework-requests* -library (see figure 10).

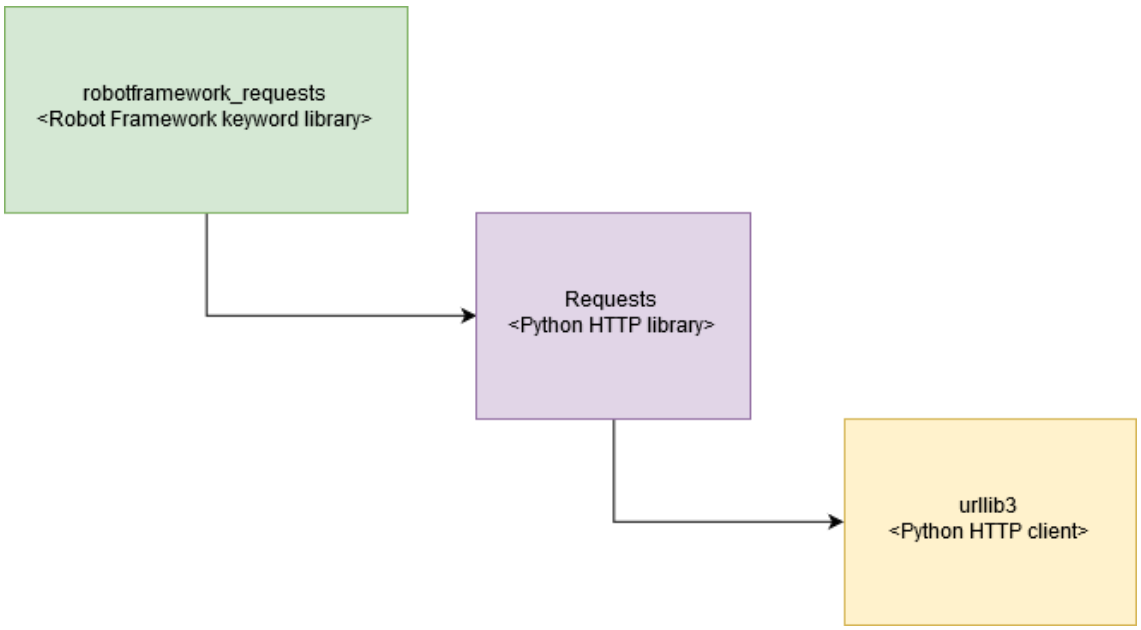

Figure 10. Underlying components used by robotframework-requests -library [27; 28.]

Investigating Robot Framework test cases revealed that maximum number of retries had been already explicitly configured to value 98. The number of retries was deemed sufficient for handling connection errors. The documentation however revealed another possible culprit for the connection errors.

Pause between retries, or *back off factor*, defaulted to value 0.1 (100 ms), as it had not been explicitly configured. This value was suspected to be too low for target devices, resulting in excessively rapid retries for the device's internal HTTP server to process. New connection errors exhibited in listing 2 were frequent in failed Robot Framework test case logs, which was deemed to support the hypothesis.

```
[ WARN ] Retrying (Retry(total=1, connect=None, read=None, redirect=None, sta-
tus=None)) after connection broken by 'NewConnectionError('<urllib3.connec-
tion.HTTPConnection object at 0x104deb350>: Failed to establish a new connec-
tion: [Errno 61] Connection refused',)'
```

Listing 2. Terminal output of a new connection error occurred during test execution.

After running multiple test suites in local and rack environments, it was verified that the increased back off factor had greatly improved test stability. The drawback of the solution was that test suite execution time had increased but was deemed acceptable for reliable test execution.

To further prevent false positives, the possibility of introducing a second test pass was explored. A secondary pass would ideally rerun only failed test cases, thus verifying that the erroneous behaviour could be reproduced. Robot Framework has built-in support for rerunning failed tests by specifying an existing XML log file and appending a *--rerunfailed* -parameter to the launch command.

Implementing rerun capability of failed test cases was straightforward, but it resulted in the creation of a new XML log file and HTML report with only records of tests run during the second pass. Combining the results was deemed necessary to publish a comprehensive report. Robot Framework's built-in Rebot-module was used to merge reports from the two passes, before publishing a final report with Jenkins Robot Framework plugin (see listing 3).

```
function rerun_robot {
    local exitcode=$?
    # rerun tests only if --rerunfailed launch parameter was given and
    # Robot exit status code indicates failed tests
    if [[ "$RERUN_FAILED_TESTS" == "true" && $exitcode -gt 0 &&
    $exitcode -le 250 ]]; then
        echo -e '\nRerunning failed tests ...\n'
        RUN_SECOND_PASS="true"
        run_robot
        echo -e '\nMerging logs ...'
        local rebot_cmd="docker-compose run test-stb rebot –outputdir
        ${OUTPUTDIR} --output output.xml --merge ${OUTPUTDIR}/output.xml
        ${OUTPUTDIR}/rerun.xml"

        eval "${rebot_cmd}"
    fi
}

function main {
    run_robot || rerun_robot
}

main "$@"
```

Listing 3. Shell script to execute *rerun_robot* -function if *run_robot* -function returns a non-zero status code.

A Docker-related aberration was discovered after lengthy testing. Performance of the Jenkins server gradually degraded after multiple days of operation, which was evident especially in UI responsiveness. Eventually the behaviour escalated to the Linux kernel executing a process that attempted to free RAM by killing processes, including Jenkins. The culprit lied within the commands used to execute test suites (see listing 4).

```
docker-compose run test rebot --outputdir ${OUTPUTDIR} --output output.xml --
merge ${OUTPUTDIR}/output.xml ${OUTPUTDIR}/rerun.xml
```

Listing 4. One of the faulty docker-compose commands used to execute a second test pass.

Fetching a list of running containers on the Jenkins server proved that each executed test suite leaked memory by leaving a running container behind. These accumulated containers ultimately utilized all RAM available on the server, leading to kernel starting the memory clean-up process. This was rectified by adding a parameter to the command, removing containers automatically on exit (see listing 5).

```
docker-compose run --rm test rebot --outputdir ${OUTPUTDIR} --output out-
put.xml --merge ${OUTPUTDIR}/output.xml ${OUTPUTDIR}/rerun.xml
```

Listing 5. Fixed docker-compose command with --rm flag added to trigger automatic removal of container on exit.

3.8 Future improvement

By the closure of this thesis' scope, the system was sufficiently reliable to be used for daily test automation work. Some options for further improvement of the system were however perceived.

Single test devices in the rack may require manual rebooting at times due to them becoming unresponsive due to unrecoverable errors. It would be beneficial to introduce a daily power-cycle reboot via a hardware timer solution. The current draw of the devices is quite low, approximately 20 W per device depending on exact model and revision, so an inexpensive digital mains timer would likely suffice for the entire rack. Implementing a power-cycle reboot would require scheduling tests not to be run during this daily maintenance period, as they would inevitably fail due to connection errors.

A test automation API for test data initialization would be advantageous, although not a part of the system per se. It would improve the level of test automation by reducing the amount of manual interaction required to initialize target devices for certain test suites relying on specific test data state. A simple concept of a test automation API use case is exhibited in figure 11.

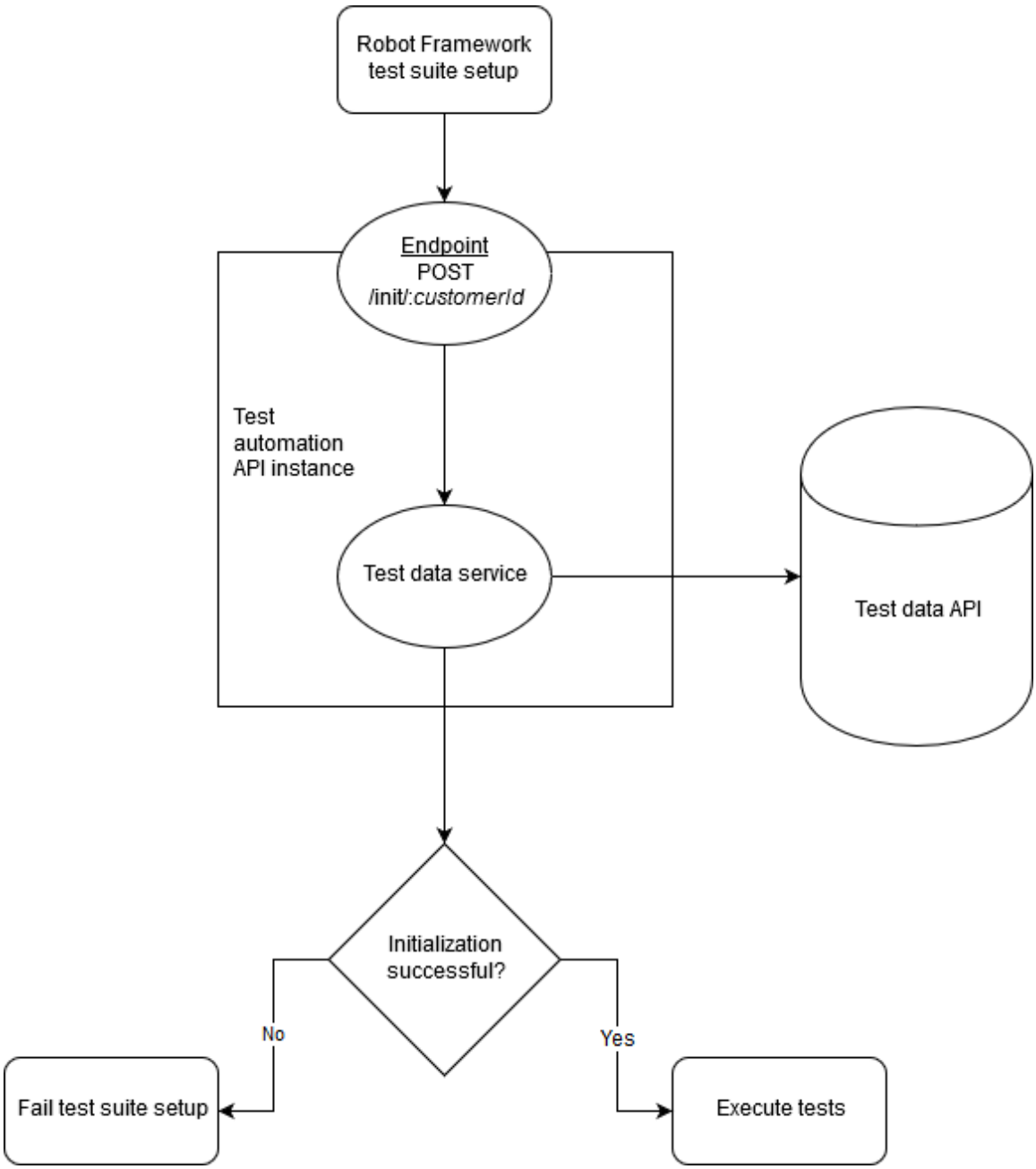

Figure 11. Concept of a rudimentary test automation API flow to initialize test data.

The amount of test devices should eventually be incremented, as it would allow even greater number of concurrent tests. The server would however require more RAM to maintain system stability when increasing the number of concurrent build executors within Jenkins. Incrementing build executors from the default value of two to four instances was experimented with, but the server's 2 GB of RAM proved inadequate for robust operation, causing unresponsiveness of the user interface.

4 Conclusion

The aim of the project documented in this Bachelor's Thesis was to fulfil newly-introduced test automation requirements. The technologies used for test automation were predefined by the management: Robot Framework as the test automation framework and Jenkins as the automation server. This software stack was fit for a multitude of tasks and deemed well-suitable for the team's endeavour as well.

The primary challenge of the project within the author's development team was to introduce a reliable automation solution for testing software modules executing on devices of a proprietary device family housed in a rack environment. The process started with verifying that test suites could be run properly in any environment. The ideal solution would require no environment-specific setup what so ever, but a brief manual setup was deemed to be inevitable to ensure information security.

After test generalization was verified to be at an acceptable level, a pool of test devices was collected, configured and installed to a device rack environment. Test device operation was verified after installation with simple network connectivity testing. The test devices were introduced as resources to the Jenkins server via Lockable Resources plugin. This ensured that devices were reserved exclusively to an automation job, as the device platform could execute a single software module at a time. Test automation jobs were created in Jenkins and the newly-configured device resources were made available for each job.

The test automation solution reached its MVP criteria at this point and it was presented to the management. The solution was tested by using it in actual software testing work, which quickly revealed some impediments in its operation. A second iteration was deemed necessary to stabilize the system for reliable day-to-day operation.

The stabilization effort began by examining network connectivity problems evident in test job debug logs. A faulty packet was captured with Wireshark network protocol analyser software and its investigation gave a hint to the root cause of the problem: the test automation server was occasionally overloading the device's internal HTTP server with requests during test execution. Additional artificial latency was introduced between request retries and the amount of networking errors greatly decreased.

A second test execution pass that re-ran only failed test cases was introduced to Jenkins test automation jobs. Executing a second pass verified that the failures could be reproduced and not caused by, for example, a brief unrelated networking error.

During the stabilization process it was also noted that Docker containers were left running after each test job and continued to consume server resources. This resulted in RAM shortage within the server environment and caused serious stability issues after lengthy uptime. The root cause was a missing parameter in the command starting containers, which would remove the container on exit. The parameter was introduced, and the change was verified to rectify the problem.

The project was accomplished successfully, and MVP criteria were met early in the roadmap timetable. The opinion was shared by both the project team and management. The test automation system was operational and used in daily software testing tasks. It was also easily expandable with more test jobs and hardware execution platforms.

The combined multidisciplinary knowledge of the team enabled the project's positive outcome. Team members agreed that carrying out the project was greatly beneficial for cross-competence within their primary development unit and thus serves future endeavours as well.

The team's approach to the project was systematic and analytical, and from a retrospective perspective, optimal. Given test automation requirements were first deconstructed to smaller subtasks. Tasks were then prioritized based on their interdependencies. This ensured an organised way of working and impacted to the project's positive outcome.

Each project phase and the overall process proved to be very educational. The project provided an opportunity to learn on theory and conventions of software testing while working on a genuinely meaningful task in a professional environment. The insight provided on software testing and test automation was professionally beneficial and complemented previous knowledge on the topic.

The motivation for writing this Bachelor's Thesis on software testing was to learn more on an important, evolving discipline within software development industry. Knowledge on testing and test automation is beneficial in any software development duties. As an example, a software developer can employ automated regression testing to reduce the

amount of manual testing. This practice can improve software quality when used as an additional safety measure between common build-time unit tests and production deployment.

References

1 ISO/IEC/IEEE 29119. Software testing. 2013. Software and systems engineering. Switzerland: IEEE.

2 Savolainen, Ossi. 2005. Ohjelmistotestaus: Testausprosessin luonti ja kehittäminen. Tietojärjestelmätieteen kandidaatin tutkielma. University of Jyväskylä. Available: <http://users.jyu.fi/~jorma/kandi/2005/Kandi_OSavolainen.pdf>.

3 Mili, Ali & Tchier, Fairouz. 2015. Software Testing: Concepts and Operations. New Jersey: John Wiley & Sons.

4 Homés, Bernard. 2011. Fundamentals of Software Testing. New Jersey: John Wiley & Sons.

5 World Quality Report 2017–18. 2017. Web document. Capgemini Group. <https://software.microfocus.com/en-us/asset/editorial-sections/world-quality-report-2017-2018>. Accessed 28 October 2017.

6 Kotilainen, Samuli. 2016. Joka päivä on testauspäivä. Tietoviikko 1.12.2016, p. 28–35.

7 Brewster, Stephen; Burrell, Darrell Norman; Dawson, Maurice & Rahim, Emad. 2010. Integrating Software Assurance into the Software Development Life Cycle. Journal of Information Systems Technology & Planning. Vol. 3, p. 49–53.

8 Taipale, Ossi. 2007. Observations on Software Testing Practice. Thesis for the degree of Doctor of Science (Technology). Lappeenranta University of Technology. Doria.

9 Isomäki, Minna; Jokela, Tero; Kaisti, Matti; Känsälä, Marja; Könnölä, Kaisa; Lehtonen, Teijo; Mäkilä, Tuomas; Rantala, Ville; Suomi, Samuli; Tuomivaara, Seppo & Ylitolva, Marko. 2014. Sulautettujen järjestelmien ketterä käsikirja. University of Turku. Doria.

10 Pan, Jiantao. 1999. Software Testing. Web document. Carnegie Mellon University. <https://users.ece.cmu.edu/~koopman/des_s99/sw_testing/>. Accessed 29 October 2017.

11 Paul, Monica. 2017. Releasing Faster - How Test Automation Can Accelerate Time to Market. Web document. uTest. <https://www.utest.com/articles/releasing-faster-how-test-automation-can-accelerate-time-to-market>. Accessed 14 January 2018.

12 Parobek, Lubos. 2016. Better software testing through automation. Web document. InfoWorld. <https://www.infoworld.com/article/3068598/application-development/better-software-testing-through-automation.html>. Updated 12 March 2016. Accessed 29 October 2017.

13 Robot Framework. Web document. Robot Framework Foundation. <http://robotframework.org>. Accessed 28 October 2017.

14 Klärck, Pekka. 2009. Writings. Web document. Eliga Oy. <http://eliga.fi/writings.html>. Accessed 28 October 2017.

15 Google Code Archive - Long-term storage for Google Code Project Hosting. Robot Framework. Web document. Google. <https://code.google.com/archive/p/robotframework/downloads?page=7>. Accessed 28 October 2017.

16 Robot Framework User Guide Version 3.0.2. 2017. Web document. Robot Framework Foundation. <http://robotframework.org/robotframework/3.0.2/RobotFrameworkUserGuide.html>. Accessed 28 October 2017.

17 Heller, Martin. 2017. What is Jenkins? The CI server explained. Web document. InfoWorld. <https://www.infoworld.com/article/3239666/devops/what-is-jenkins-the-ci-server-explained.html>. Accessed 2. January 2018.

18 Build a Python app with PyInstaller. 2017. Web document. Jenkins. <https://jenkins.io/doc/tutorials/build-a-python-app-with-pyinstaller/>. Accessed 2 January 2018.

19 On Elisa. 2017. Web document. Elisa Oyj. <http://corporate.elisa.com/on-elisa/>. Accessed 21 November 2017.

20 Organisation. 2017. Web document. Elisa Oyj. <http://corporate.elisa.com/on-elisa/organisation/>. Accessed 21 November 2017.

21 What is a Container. 2017. Web document. Docker. <https://www.docker.com/what-container>. Accessed 2 December 2017.

22 Kern, Alexander. 2016. [JENKINS-35628] java.lang.IllegalStateException on any unlock. Web document. Jenkins JIRA. <https://issues.jenkins-ci.org/browse/JENKINS-35628>. Accessed 21 November 2017.

23 Robot Framework Plugin - Jenkins. 2017. Web document. Jenkins Confluence. <https://wiki.jenkins.io/display/JENKINS/Robot+Framework+Plugin>. Accessed 27 November 2017.

24 Petrov, Andrey. 2017. Reference - urllib3 dev documentation. Web document. <https://urllib3.readthedocs.io/en/latest/reference/index.html>. Accessed 28 December 2017.

25 Berners-Lee, Tim; Fielding, Roy & Frystyk, Henrik. 1996. Hypertext Transfer Protocol – HTTP/1.0. Web document. HTTP Working Group. <https://www.w3.org/Protocols/HTTP/1.0/spec.html>. Accessed 29 December 2017.

26 Berners-Lee, Tim; Fielding, Roy; Frystyk, Henrik; Gettys, Jim; Leach, Paul; Masinter, Larry & Mogul, Jeffrey. 1999. Hypertext Transfer Protocol – HTTP/1.1. Web document. The Internet Society. <https://www.w3.org/Protocols/rfc2616/rfc2616.html>. Accessed 29 December 2017.

27 GitHub – requests/requests: Python HTTP Requests for Humans. 2017. Web document. GitHub. <https://github.com/requests/requests>. Accessed 30 December 2017.

28 RequestsKeywords. 2017. Web document. <http://bulkan.github.io/robotframework-requests/>. Accessed 30 December 2017.